AN INTERPROCESSOR COMMUNICATION PROTOCOL WITH SMART STREAMING PORT

TECHNICAL FIELD

This invention relates in general to the field of electronics, and more specifically to an InterProcessor Communication (IPC) protocol/network with a smart streaming port.

BACKGROUND

Most electronic systems include a number of networked elements (components) such as hardware and software that form the system. In most systems there is a layer responsible for communication between the different components that form a networked element as well as between the different networked elements themselves. This layer is typically referred to as the InterProcessor Communication (IPC) layer.

Several protocols have been introduced in the last few years to deal with interprocessor communications. One example of an IPC product is PCI AGP Controller (PAC) that integrates a Host-to-PCI bridge, Dynamic Random Access Memory (DRAM) controller and data path and an Accelerated Graphics Port (AGP) interface. Another example of an IPC product is the OMAP™ platforms. Neither of these platforms provides much if any support above the hardware level and provides little design flexibility at the lower level component or channel levels (physical layer).

The PAC platforms for example, are closed architectures and are embedded into the Operating System's TAPI layer, with the IPC code not being accessible to developers. Therefore, these platforms do not extend to the component levels they

also do not allow for dynamic assignment of IPC resources, hardware support capabilities, or multi-node routing.

One problem with applications such as real-time processing applications is the discovery of services on different processors and the requirement for quick processing of the IPC requests. One aspect of guaranteeing good support in these situations is to offset the work of the different applications such as Mobile Applications (MAs) in a radio communication device, and provide for co-processing support prior to delivery of the data to the target MA. This co-processing support typically requires smart hardware ports that can transport the IPC data to the co-processor and then route the data back to the target MA. Another aspect of guaranteeing good support is by using dedicated links (ports) to carry specific software services to software components on the different MAs. Today, these methods are neither dynamic nor runtime configurable. Given the above, a need thus exists in the art for an IPC protocol with a smart streaming port that can provide a solution to some of these shortcomings in the prior art.

BRIEF DESCRIPTION OF THE DRAWINGS

The features of the present invention, which are believed to be novel, are set forth with particularity in the appended claims. The invention may best be understood by reference to the following description, taken in conjunction with the accompanying drawings, in the several figures of which like reference numerals identify like elements, and in which:

FIG. 1 shows a diagram of an IPC network in accordance with an embodiment of the invention.

FIG. 2 shows an IPC stack in accordance with an embodiment of the invention.

FIG. 3 shows an IPC component IPC assignment in accordance with an embodiment of the invention.

FIG. 4 shows the main IPC tables in accordance with an embodiment of the invention.

FIG. 5 shows a diagram showing channel allocation in accordance with an embodiment of the invention.

FIG. 6 shows a diagram highlighting the steps involved during an IPC client initialization routine in accordance with an embodiment of the invention.

FIG. 7 shows another diagram highlighting the steps involved during an IPC client initialization in accordance with an embodiment of the invention.

FIG. 8 shows a diagram highlighting the first level of IPC encapsulation in accordance with an embodiment of the invention.

FIG. 9 shows a diagram highlighting the steps taken during IPC component initialization in accordance with an embodiment of the invention.

FIG. 10 shows a chart highlighting the steps taken during component initialization in accordance with an embodiment of the invention.

FIG. 11 shows the transfer of IPC data between an IPC client and an IPC server in accordance with an embodiment of the invention.

FIG. 12 shows a diagram of an IPC data header in accordance with an embodiment of the invention.

FIG. 13 shows a diagram of the steps taken during an IPC data request in accordance with an embodiment of the invention.

FIG. 14 shows an IPC network in accordance with an embodiment of the invention.

FIG. 15 shows an electronic device such as a radio communication device in accordance with an embodiment of the invention.

FIG. 16 shows an IPC protocol with dedicated port(s) in accordance with an embodiment of the invention.

FIG. 17 shows the steps taken during a request to dedicate a smart hardware port in accordance with an embodiment of the invention.

FIG. 18 shows a continuation of FIG. 17 highlighting the transfer of data once the ports have been dedicated in accordance with an embodiment of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

While the specification concludes with claims defining the features of the invention that are regarded as novel, it is believed that the invention will be better understood from a consideration of the following description in conjunction with the drawing figures.

In accordance with the present invention, port assignment to specific services is dynamic, abstracted from the software components and can be used to provide more than one service (e.g., combination of audio and/or video on one link). The advantages of the IPC to dynamically link a port to a service in accordance with an embodiment of the invention can be summarized as follows:

1). Alleviates latency issues for software components when traffic becomes a problem on a non-dedicated IPC link.

2). Provides for co-processing support on the hardware port.

3). Component architecture abstraction from platforms.

4). Component portability between different Mobile Applications (MAs).

The IPC provides the support needed for different processors operating in a system to communicate with each other. For example, in dual-processor (or multi-processor) radio architecture for use in a radio communication device that includes an

Application Processor (AP) and a Baseband Processor (BP), the IPC provides the support needed for the processors to communicate with each other in an efficient manner. The IPC provides this support without imposing any constrains on the design of the AP or BP.

The IPC allows any processor that adopts the IPC as its inter-processor communication stack to co-exist together and operate as if the two were actually running on the same processor core sharing a common operating system and memory. With the use of multiple processors in communication devices becoming the norm, the IPC provides for reliable communications between the different processors.

The IPC hardware provides the physical connection that ties together the different processors to the IPC network. In one embodiment of the invention, data packets are preferably transported between the different hosts asynchronously. Processors that are connected to the IPC network have their physical and logical addresses statistically or dynamically assigned (e.g., IPC addresses). Also, since data packets can flow in any direction within the IPC network in one embodiment of the invention, to the packets carry a destination address of the processor that they are trying to reach. Packets are also preferably checked for errors using conventional Cyclic Redundancy Check (CRC) techniques. Although the network activities of the IPC network of the present invention may have some similarities to those found on an Internet network that uses IP transport layers such as a Transmission Control Protocol/Internet Protocol (TCP/IP) network, the IPC of the present invention is not divided into smaller networks with gateways as in a TCP/IP network.

Referring now to FIG. 1, there is shown an IPC network 100 in accordance with an embodiment of the invention. The IPC network 100 includes a plurality of IPC clients 102-106, and an IPC server 108 coupled to the IPC clients 102-106 using different IPC physical links such as shared memory 110, Universal Asynchronous

Receiver/Transmitter (UART) 112 and Universal Serial Bus (USB) 114 as some illustrative examples. It should be noted that with the IPC of the present invention, an IPC client 102-106 can negotiate with the current IPC server 108 to switch roles. If an IPC client 102-106 negotiates to become the IPC server and becomes the new IPC server, all of the remaining IPC clients are instructed to change the IP address of the server given the change in the IPC server.

In FIG. 2, there is shown an IPC stack 200 of an IPC server 108 (or IPC clients 102-108) in accordance with an embodiment the present invention. The IPC stack 200 is designed to be integrated under an Operating System (OS) and to provide support for the inter-processor communication needs of component traffic. The IPC stack is composed of the following 3 main layers:

(1). **IPC Presentation Manager (202)** – this layer is used to translate different data types between different system components (e.g., software threads).

(2). **IPC Session Manager (204)** – this layer is a central repository for all incoming/outgoing IPC traffic between the IPC stack and all of the system components. The IPC session manager 204 has several functions: assignment of component IDs for participating IPC components; deciding if the IPC data needs to be encapsulated; routing of IPC data, termination of IPC traffic; place holder for IPC processors; providing IPC addresses, assigning and authenticating IPC clients, etc.

IPC Transport Layer (208) – located within the IPC session manager (layer) 204, the IPC transport layer 208 provides a very basic cyclic redundancy check for the purpose of transporting the IPC data between the different processors. In addition, the IPC transport layer 208 is responsible for routing IPC messages to their final destinations on the IPC network 100. The routing function of the transport layer is enabled only on IPC servers.

IPC Router Block (210) – transports the IPC data to a destination component (not shown). Incoming IPC messages carry among other things, the originator component ID, the IPC message opcodes such as Audio and Modem. Note that in accordance with an embodiment of the invention, a unique opcode is assigned to each component/software thread (see for example 502 in FIG. 5), such as Audio and Modem that is coupled to the IPC network. The IPC session manager 204 relies on the router block 210 to send the IPC data to the right component(s).

(3). Device Interface Layer (206) - is responsible for managing the IPC physical-to-logical IPC channels. Its main function is to abstract the IPC hardware completely so that the stack IPC becomes hardware independent. The device interface layer 206 manages the physical bandwidth of the IPC link underneath to support all of the IPC logical channels. In the incoming path, the device interface layer 206 picks up data from different physical channels 110-114 and passes them up to the rest of the IPC stack. On the outgoing path, the device interface layer 206 manages the data loading of the IPC logical channels by sending them onto the appropriate physical channels. The device interface layer 206 also handles concatenating IPC packets belonging to the same IPC channel before sending them to the IPC hardware. Channel requirements are pre-negotiated between the IPC session manager 204 and the IPC device interface layer 206. The device interface layer 206 provides for hardware ports, which in turn provide a device interface to an IPC client 102-106.

Referring to FIG. 3 there is shown an IPC component ID assignment routine. Any new component wishing to participate in an IPC communication must do so by first requesting an IPC Identification Number (ID) in step 302 from its IPC session manager (e.g., like session manager 204). The local session manager (e.g., session manager located in client that the component is coupled to) will then alert the IPC

server session manager of the new IPC components and a component ID assignment will be provided in step 304. In accordance with an embodiment of the invention, the component IDs are dynamic and can be reassigned by the session manager. The main IPC server location will most likely be on the main AP. Each IPC node will preferably have a unique IPC node ID and the session manager will keep in its database the following information for each participating IPC node:

- IPC Node Type: For example, a particular BP or AP, a Wireless Local Area Network (WLAN) AP, etc.
- IPC address: The IPC address of the IPC node.
- Data Type: The data type of the IPC node.
- Opcode list: This is a list of all the IPC message opcodes that the components have subscribed to.
- Component IDs: List of all the component IDs.

Referring now to FIG. 4, there is shown an IPC stack along with all of the main IPC tables. The Dynamic routing table 402 includes the Node Type, IPC address/Port # information, Data Type and Subscription list. The component routing table 404 includes the information linking the Opcode information and all of the components subscribed to each particular Opcode. Finally, the Channel Resource table 406 includes a linking of each Channel ID with a list of physical channel IDs.

In FIG. 5, there is shown a block diagram of how the IPC stack in accordance with an embodiment of the invention, provides an IPC channel for a component such as a software thread (e.g., Audio, etc.). Component 502 first requests an IPC channel in step 504. The session manager shown in FIG. 5, negotiates the component's request with the Device Layer in step 506 using a defined API. The Device layer (Device Interface) then requests hardware resources, such as a data channel 508. The

session manager shown in FIG. 5 in response to the request, grants an IPC channel to the requester in step 510. The component 502 next sends its data on the assigned channel 508. The device layer then forwards the data to the IPC network. The mapping of the logical to physical channel IDs is the function of the IPC device interface.

Referring now to FIG. 6, the first step in IPC client initialization is sending a registration request (step 606) between the IPC client 602 and the IPC server 604. The IPC server 604 then authenticates the request with the IPC client 602 in step 608. This is followed by sending an IPC address to the IPC client and completing the registration in step 610. The IPC client's session manger sends a copy of its dynamic routing table to the IPC server in step 612.

More detailed steps taken during the IPC client initialization process are shown in FIG. 7. The client session manager (shown in table as Session (client)) sends a configuration request to the IPC server's session manager (shown in table as Session (Server)) in step 702. In step 704, authentication is requested by the IPC server's session manager. Authentication between the IPC client and IPC server is then carried out in step 706.

The parameters in the configuration request include the node type and the data type. The session server in response to the configuration request in step 702 assigns the requestor an IPC address. It also sets up a dynamic routing table for the requestor if one does not exist. It then sends the requestor a configuration indication as in step 708. The configuration indication parameters include the IPC address of the server and the newly assigned IPC address of the client.

In response to receiving the configuration indication, components attached to the session client can request control/data from the client's session manager. The Session client then sends a configuration indication confirm message to the session

server in step 710. The "configuration indication confirm" message has no parameters, upon receiving the configuration indication confirm message in step 710, and the session server can initiate IPC streams to the newly configured session client. The session server then sends configuration update messages to the session clients in steps 712 and 714. This causes the both session clients shown in FIG. 7 to update their respective dynamic routing tables (not shown) and send a configuration update confirm message to the session server in steps 716 and 718. Upon receiving the configuration update confirm messages, the session server makes sure all of the IPC participants have been updated.

When a packet is received by an IPC session manager, it comes in the form of data that includes the source component ID, the destination ID, a channel ID and the type of BP or AP. The IPC session manager will add the destination component ID in the event that the destination ID is not inserted. The IPC session manager will also insert an IPC address. It is the IPC session manager that discovers the destination ID based on the message opcode received. The destination ID is based on a lookup table. This lookup table is updated dynamically each time a component subscribes to a new IPC message opcode (e.g., an audio component subscribes to audio messages by sending a request to the IPC session manager).

In FIG. 8 there is shown a sequence of events during a general destination ID discovery sequence between a component and its IPC session manager in accordance with an embodiment of the invention. In step 802, the component sends its source ID (but no destination ID), the type of the destination BP or AP and the IPC data which includes a header and data. In step 804, the IPC session manager looks at the IPC data header opcode and the type of destination BP or AP, in order to lookup the corresponding dynamic routing table and find the correct destination address. In step

806, the IPC session manager inserts the IPC address of the component and sends it down to the device layer.

In FIG. 9, typical steps taken during an IPC component initialization are shown. Once the BP has been configured by the IPC server shown in FIG. 9, it allows components such as component 902 to subscribe to different services. Components will subscribe themselves to functions such as Audio, Video, etc. in step 904. The component subscription information is then sent to the IPC session manager for component ID creations (if an ID is not assigned yet) and creation or updating of the dynamic routing table for a particular IPC address (step 906). In step 908, the session manager updates the IPC server with the information from step 906. An confirmation of the update is sent by the IPC server to the IPC client in step 912 upon receiving the updated dynamic routing table in step 908. Once the server is alerted, new dynamic routing table updates are broadcast to all participating processors in step 910.

The same component initialization process is shown between a component (client) 1002, a session (client) also known as a client session manager 1004 and the session (server) also known as the server session manager 1006 in FIG. 10. A component configuration request in step 1008 is sent by the component (client) 1002. In response to the request, the client session manager 1004 negotiates a logical channel with its device layer (not shown). The client session manager 1004 also assigns a component ID and adds the new opcode list to its dynamic routing table (not shown). In step 1010, the client session manager 1004 sends a configuration reply which includes a component ID and a channel ID as parameters. In response to the configuration reply, the component (client) 1002 receives its ID and channel ID from the client's session manager 1004.

Once the client session manager 1004 replies in step 1010 to the configuration request in step 1008, the client session manager 1004 sends a configuration update

request in step 1012 to the session server (server session manager) 1006. The parameters for the configuration update request are any new changes that have been made in the dynamic routing table (not shown). The server's session manager 1006 updates the dynamic routing table for that IPC address. The server's session manager 1006 in step 1016 then sends all the IPC clients (not shown) a configuration update, while it sends the client's session manager 1004 a configuration update indication in step 1014. The server's session manager 1006 makes sure the server has updated its routing table with the changes that were sent.

In the configuration update message of step 1016 which includes the dynamic routing tables as a parameter(s), the session server 1006 updates the dynamic routing tables and sends a configuration update confirm message in step 1018. The server then makes sure all of the IPC participants have been updated.

The IPC session manager determines the routing path of incoming and outgoing IPC packets. The route of an outgoing packet is determined by the component's IPC address. If the destination address is found to be that of a local processor, a mapping of the IPC to the Operating System (OS) is carried out within the session manager. If the destination address is found to be for a local IPC client, the packet is sent to the IPC stack for further processing (e.g., encapsulation). Note that if the destination component is located on the same processor as the component sending the IPC packet, no encapsulation is required and the packet gets passed over through the normal OS message calling (e.g., Microsoft Message Queue, etc.). In this way components do not have to worry about modifying their message input schemes. They only need to change their message posting methodologies from an OS specific design to an IPC call instead.

For incoming packets, if the destination address of the message is not equal to the IPC server's, the incoming packets gets routed to the proper IPC client. The

routing of incoming packets is handled by the IPC server's session manager. Otherwise, the message is forwarded to the right component or components depending on whether or not the component destination ID is set to a valid component ID or to 0XFF.

The IPC router block (e.g., see IPC router block 210 in FIG. 2) transports the IPC data to the destination component. Incoming IPC messages carry among other things, the originator component ID and the IPC message opcodes such as those for Audio, Modem, etc. The IPC session manager relies on its component routing table to send the IPC data to the right component(s). Both the dynamic routing table and the component routing table are updated by the IPC server/client.

During power-up, each component must register itself with its session manager to obtain an IPC component ID. In addition, it must also subscribe to incoming IPC messages such as Audio, Modem, etc. This information is stored in the component routing table for use by the IPC session manager.

When a component (e.g., software thread) 1102, as shown in FIG. 11, sends its data request to the IPC session manager as in step 1104, a check is made on the destination IPC node (e.g., the BP). If the IPC node does not support the IPC message opcode, an error reply is returned to the component 1102. In addition to the error reply, the IPC session manager returns an update of all the IPC nodes that are capable of receiving that particular opcode. It is up to the component to decide to which of the IPC node(s) it will redirect the message. The IPC session manager 1106 will proceed to encapsulate the data with the IPC header information before the data is sent on the IPC network if the session manager determines that the destination component is located in the IPC network but not in the local processor.

In FIG. 12, there is shown an IPC data header 1202 in accordance with an embodiment of the invention. The header includes the source and destination IPC

addresses, source port, destination port provided by the IPC router, the Length and checksum information provided by the IPC transport and the source IPC component and Destination IPC component provided by the session manager. The Message opcode, message length and IPC data are provided by the component 1204.

A typical IPC data request in accordance with an embodiment of the invention is shown in FIG. 13. In step 1302, a component sends an update request. The component update parameters preferably include a node type and opcode. The component searches for Node types that support its destination opcode. If the Node type is equal to 0xFF, a session manager (labeled as "session" in FIG. 13) proceeds to send the component information to all the node tables for all IPC participants. If the opcode field is equal to 0xFF, the session manager shown in FIG. 13 proceeds to send the component the opcode list belonging to the specified Node type. On the other hand, if the opcode has a specific value, the session manager proceeds to send the component a true or false value corresponding to whether the Node type supports or does not support that particular opcode.

In step 1304, the component update indication is sent to the component. If the node type is equal to 0xFF, the node tables are returned to the component. If the opcode field is equal to 0xFF, the list of opcodes is returned to the component. However, if the opcode is a specific value, a true or false message is returned. In step 1306, a component data request is made. The parameters for the component data request include the node type, the IPC message opcode, the IPC message data, the channel ID and the component ID. In a component data request, the session manager checks the node type to determine whether the opcode is supported. If the node type does not support the opcode, a component update indication is sent in step 1308. If however, the node type supports the opcode, a data request is sent to the device layer

in step 1310. The data request parameters include the IPC message, the channel ID and the IPC header.

The device layer schedules to send the data request message based on the channel ID. The device layer selects the IPC hardware based on the port # header information. Once the data is committed, a data confirm message is sent to the session manager in 1312. In step 1314, the session manager proceeds to send a component data confirm message to the component. The component can wait for the confirmation before sending more IPC messages. Once a data confirm is received, the component can proceed to send the next IPC message.

In step 1316, the device layer sends a data indication message including IPC message data and an IPC header. The session manager checks the destination IPC header of the message, and if different from the local IPC address, the session manager sends (routes) the message to the right IPC node. In step 1310, the session manager sends a data request to the device layer with a reserved channel ID. The session manager checks the destination component ID, and if it is equal to 0xFF, routes the message to all the components subscribed to that opcode. In step 1318, the session manager sends a component data indication message and the component receives the IPC data.

The IPC stack uses a reserved control channel for communication purposes between all participating IPC nodes. On power-up, the IPC server's session manager uses this link to broadcast messages to IPC clients and vice versa. During normal operations, this control channel is used to carry control information between all APs and BPs.

In FIG. 14, there is shown the control channels 1402-1406 located between the IPC stacks (labeled as IPC stack and IPC stack server) and the IPC hardware. Control channel information 1408 is also transmitted along with data packets 1410 when

sending data between different IPC hardware. An IPC client broadcasts its configuration request initially on the IPC control channel. The IPC stack server receives the broadcast and responds with an IPC address for that client. This IPC address becomes associated with the dynamic routing table for that particular processor (AP or BP).

IPC APPLICATION PROGRAM INTERFACES (APIs)

Below are listed some of the APIs for the IPC protocol of the present invention.

1). Component Interface to the IPC session manager:

CreateComponentInst()
Creates a component database in the IPC session manager. Information such as component data types (Big Endian vs. little Endian) and subscription to message opcodes are used in the dynamic data routing table belonging to an IPC address.

OpenChannelKeep()
Open an IPC channel and if one is available, a ChannelGrant() is issued. The channel is reserved until a CloseChannel() is issued. Components send QoS requests to the IPC session Manager. The IPC channel assigns a component ID if one is not yet assigned (e.g.ChannelGrant()).

OpenChannel()
Open an IPC channel and if one is available, a ChannelGrant() is issued. The parameters are the same used for the OpenChannelKeep() primitive.

OpenChannelWThru()
Open an IPC channel and if one is available, a ChannelGrant() is issued. This is a request for a write thru channel signifying that encapsulation be turned off on this channel (e.g. Non UDP AT commands).

CloseChannel()
Request that an IPC channel be closed. The Component no longer needs the channel. The resources are then freed.

ChannelGrant()
A channel is granted to the requestor. The Channel IDs are assigned by the IPC session manager if one is not yet assigned.

ChannelError()

A channel error has occurred. The channel is closed and the requestor is notified.

ChannelDataIndication()
The requestor is alerted that data on a channel is to be delivered. This message is sent by the IPC presentation manager to the target component. This also includes control channel data.

DataChannelRequest()
The requestor wants to send data on an opened channel. This also includes control channel data.

ChannelClose()
Request that an IPC channel be closed. A channel inactivity timer expired and the Channel associated with the timeout is closed. This could also be due to channel error.

2). IPC session manager to/from IPC device interface

OpenChannel()
Open a logical IPC channel and if one is available, a ChannelGrant() is issued. The IPC session manager sends channel priority requests to the IPC device interface manager.

CloseChannel()
Request that an IPC logical channel be closed. A component decides that it no longer requires the channel.

ChannelGrant()
A logical channel is granted to the requestor.

ChannelError()
A channel error has occurred (e.g. CRC failure on incoming data or physical channel failure).

ChannelDataIndication()
The requestor is alerted that data on a channel is to be delivered.

DataChannelRequest()
The requestor wants to send data on the logical channel.

ChannelClose()
Request that an IPC channel be closed. A channel inactivity timer expired and the Channel associated with the timeout is closed. This could also be due to channel error.

3). IPC session manager to IPC presentation manager

ChannelDataIndication()
The requestor is alerted that data on a channel is to be delivered. The information is to be forwarded to the target component with the correct data format.

4). IPC Hardware/IPC Stack Interface

OpenChannel()
Open a physical IPC channel and if one is available, a ChannelGrant() is issued. The IPC session manager sends channel priority requests to the IPC Hardware.

CloseChannel()
Request that an IPC physical channel be closed. The component no longer requires the channel.

ChannelGrant()
A physical channel is granted to the requestor.

ChannelError()
A channel error has occurred (e.g. CRC failure on incoming data or physical channel failure).

ChannelDataIndication()
The requestor is alerted that data on a channel is to be delivered.

DataChannelRequest()
The requestor wants to send data on the physical channel.

ChannelClose()
Request that an IPC channel be closed. A channel inactivity timer expired and the Channel associated with the timeout is closed. This could also be due to channel error.

In FIG. 15, there is shown a block diagram of an electronic device such as a radio communication device (e.g., cellular telephone, etc.) 1500 having a baseband processor (BP) 1502 and an application processor (AP) 1504 communicating with each other using an IPC network. The IPC protocol of the present invention provides for communications between multiple processors in a system such as a communication device 1500. The IPC allows for a Mobile Application (MA) client (e.g., iDEN™ WLAN) to register with a MA server such as a Personal

Communication System (PCS) application, and will provide the means for the two MAs to communicate freely without any limitations on what software architectures, operating systems, hardware, etc. each depend on within its own MA.

The IPC protocol allows for the dynamic addition of any IPC conforming MA into the IPC link for communication. Thus, an IPC network is formed without any compile time dependencies, or any other software assumptions. The IPC of the present invention presents a standard way for software components to communicate with the IPC stack (not shown in FIG. 15) and the hardware below the stack is also abstracted such that components can choose different links to communicate.

SMART STREAMING PORT

In accordance with one embodiment of the invention, hardware ports that are used by the IPC stack to transport the IPC data can dynamically be assigned to a co-processing function such as the summing of two audio streams together before sending the combined stream out on the IPC network. These functions provide significant support for multi-media, gaming, etc. Other potential uses include mixing, synchronizing and rate conversion operations to name just a few. Operations performed on the data streams can make the combined data very multi-media attractive without taking up more bandwidth. Things like producing stereo audio, adding voice on top of the background music for gaming, etc. all can be easily implemented using this IPC with smart streaming ports.

In addition to co-processing, the IPC session manager may find at any point that the traffic loads on a particular IPC link are rather high which may be delaying the delivery of the data to a component on time. The discovery of the latency can be either an IPC functionality or a component requesting a faster delivery of its awaited data. The idea stems from the IPC architecture which provides a way for its session

manger to decide which port it wants a particular message to flow in and out of. The IPC session manager ties a list of opcodes to a port such as a Synchronous Serial Interface (SSI) port, although it should be noted that any other transport mechanism supported by the IPC can be used.

As an exemplary illustration, assuming that a particular application MA that is on either a dedicated link or on a non-dedicated IPC link, detects that the non-dedicated link is not providing adequate real-time delivery of the data it needs to send, both MAs can negotiate the use of a dedicated link instead. In this illustrative case, an SSI port is the likely solution to the problem, assuming that it is available.

In accordance with the present invention, an IPC session manager (either in a client or a server) has among other things the intelligence to route messages and manage the distribution of the messages. The IPC session manager creates and maintains dynamic routing tables to discover the IPC nodes capabilities as well as figure out the routing paths of the IPC messages. Although the session manager can only tell that a port is available, it discovers its capability through using an API between the session manager and the device layer. The device layer knows exactly the capabilities for each of the ports found in the layer.

If for example, the IPC session manager decides to dedicate a port to a particular service, it will create a link for all opcodes that belong to such a service to a port (e.g., SSI). From then on, every IPC message that carries a particular opcode belonging to that port will get routed as such. In addition to dedication of the port, the port is called upon to perform co-processing functions on the data stream per the IPC channel. A command header is sent with the data to the port requesting certain co-processing functions to be performed. For example, an audio stream from one channel may be requested to rate convert to a certain rate, or to sum itself with other streams, etc. The output of the port is one stream integrating multiple streams

together and is now ready to be sent to the target MA. As an example, the SSI would be dedicated to carry audio as is done today with different platforms. The IPC session manager will forward its request to the device layer with the port information so that it can be delivered specifically on it. Each channel will carry at the beginning of its data a co-processing command block which will specify port operation on that channel stream.

Referring to Fig. 16, there are shown three software threads 1602, 1604 and 1606 each having different bandwidth requirements coupled to an IPC stack 1610 having a session manager 1608 in accordance with the invention. An SSI port 1610 has been assigned (dedicated) to receive audio data from the software threads; here software threads 1602 and 1604 are shown sending their data through SSI port 1610 located in device layer 1612.

In FIG. 17, there is shown a diagram highlighting the steps taken in dedicating a port in accordance with an embodiment of the invention. In step 1702, a request from a software thread 1704 is made for a port (e.g., SSI port 1714) to be dedicated). For example, SSI port 1714 may be dedicated for a service such as audio. Dedicating a port means that only IPC data is sent on that port between clients and there is no IPC header overhead. For example, the SSI port 1714 can be dedicated to carry voice samples between IPC A client and IPC B client. In this example, since the port has been dedicated, clients A and B do not have to send IPC headers along with their data, thereby reducing system overhead.

In step 1704, the list of op-codes dedicated to the ports is reviewed by the session manager 1712. In step 1706, session manager 1712 looks for conflicts between those op-codes previously dedicated to the port in question. If the session manager finds that a particular port is already used, it can renegotiate the use of the port with the other processor. If the session manager finds no conflicts, it can

dedicate the port to certain opcodes and IPC clients. In step 1708, the session manager 1712 dedicates the use of the SSI port 1714 to audio service. In addition to handling a type of service like audio, SSI port 1714 may be asked to perform co-processing on the data stream per the IPC channel. In step 1710, the session manager 1712 sends control information to the SSI port 1714 informing it of what co-processing functions to perform on the data stream. In the example shown in FIG. 17, the audio streams sent by software thread 1716 may be asked to be combined with the audio stream sent by software thread 1718 to SSI port 1714.

In Fig. 18, there is shown the same audio streams sent by software threads 1716 and 1718. The session manager 1712 adds a command header 1802 and 1804 to each steam of data sent by software threads 1716 and 1718. The command header 1802 and 1804 will instruct SSI port 1714 to perform a certain co-processing to be performed to the data it receives. As an example, the command header 1802 may ask SSI port 1714 to rate convert the data to a certain rate; sum the data with other streams of data, etc. In this particular example, the output stream of data from SSI port 1714 is one stream integrating the steams from software threads 1716 and 1718.

As an illustrative implementation example, an application/component requests a type of "Service" (e.g., rate conversion, etc.) from the session manager 1712. The session manager 1712 negotiates with the device layer for that particular type of "Service". The negotiation for service entails a request that is made up of a request ID which lets the device layer what type of service is being requested, length of request and data parameters. The device layer grants or denies such service based on the availability of hardware (e.g., smart ports, etc.) or other resources that are needed to support the service. If the device layer grants the request for the Service, the device layer grants a SERVICE ID for that service. The SERVICE ID describes the service requested on that channel from the device layer. The SERVICE ID can be just a

number. The session manager 1712 forwards the SERVICE ID to the component that requested the service. The component will then send the SERVICE ID in the co-processor command block when it wants to have that service performed in that channel. The component will not however use the SERVICE ID in conjunction with the channel ID if it wishes to have a service associated with that channel.

By linking a port to a service as done in the present invention, many advantages are achieved, including allowing for co-processing support to be performed at the hardware port, component architecture abstraction from platforms, and reduction in latency issues when traffic becomes a problem on a non-dedicated IPC link, etc. The session manager 1712 can make the decision to dedicate a port on its own if for example it determines that there is too much traffic on one port, or it can receive a request from a component.

While the preferred embodiments of the invention have been illustrated and described, it will be clear that the invention is not so limited. Numerous modifications, changes, variations, substitutions and equivalents will occur to those skilled in the art without departing from the spirit and scope of the present invention as defined by the appended claims.

What is claimed is: